REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
May 1990	Conference paper	

4. TITLE AND SUBTITLE

See report.

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

See report.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

See report.

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

See report.

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A - Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Presented at the IEEE 1990 National Aerospace and Electronics Conference (NAECON 1990) held in Dayton, Ohio, on 21-25 May 1990.

14. ABSTRACT

See report.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU		**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified			

EXPLANATION GENERATION IN EXPERT SYSTEMS

Capt Vance M. Saunders
WRDC/AAWA-1
WPAFB, Ohio

Verlynda S. Dobbs
Wright State University
Dayton, Ohio

Abstract

Today's technology provides tremendous amounts of information at incredible speeds. In order to make this information useful for more complex, significant problem solving applications, intelligent computer software systems are needed. The Expert System (ES) technology of Artificial Intelligence (AI) is one solution that is emerging to meet this need. However, as this technology continues to develop and as we begin to use expert machines more and more, it is crucial that we demand the same explanatory capability from these mechanical experts as we do from human experts.

This paper examines the Explanation Facilities (EFs) of ESs by presenting some background information on explanation generation and by discussing the development of a specific EF.

1 Introduction

The purpose of this paper is to examine the *explanation generation capabilities* of expert systems. As the technology of building *computerized experts* transitions more and more from the academic laboratory to different operational, on-line environments (business, industry, defense, etc.), the importance of incorporating software engineering practices into the development of these ESs needs to be reemphasized. Due to the complexity and seriousness of the problems ESs are being developed to solve, **we simply can't afford the cost associated with producing systems that are unreliable or produce incorrect results.**

This paper begins by presenting some general background information on explanation generation in ESs and then moves into a discussion of an **E**xplanation **F**acility for a **F**rame-based **E**xpert **S**ystem **S**hell (EFFESS). Topics presented in this discussion include EFFESS's scope, underlying knowledge representation structure (KRS), design, and functionality.

2 Background

Definition And Importance

The Random House College Dictionary defines *explanation* as:

> "to make plain, clear, or intelligible something that is not known or understood". [13]

Chandrasekaran et al. [4] identifies two different types of explanation that this general definition encompasses with respect to ESs. These two types of explanation are *explaining the world* (explaining objects and processes external to an ES) and *explaining decisions* (explaining an ES's own internal objects and processes). It is this internal type of explanation that we are interested in in this paper.

While some of the first ESs did not have EFs, it didn't take long for their need to be identified. Today, EFs are considered as important to an ES as its knowledge base, control strategy, or inferencing mechanism. In fact, Firebaugh [8] emphasizes that it is **the explanation facility of an ES** that distinguishes it from other knowledge-based AI programs. In other words, **an expert system without an explanation facility is not an expert system.** Regardless of whether one agrees with Firebaugh or not, few individuals (if any) working in ES research and development will argue against the need for EFs. There are three major reasons why this is true.

The first reason EFs are important stems from the rationale requiring an expert to be able to explain himself. Schank [11] points out that we humans don't allow other human beings to come up with new ideas, concepts, etc. unless these new ideas and concepts can be explained to us in some understandable way. In other words, experts must be able to explain themselves because we humans require it in order to have confidence in what they are telling us. EFs satisfy this same requirement for ESs.

While this first reason argues that we humans **won't** accept ESs that lack an explanation facility, reason number three argues that we **can't** accept ESs that lack an explanation facility. Says Forsyth,

> "The explanation facility should not be regarded as an optional extra. Donald Michie (1982) and others have warned about the dire consequences of systems which do not operate within the 'human cognitive window', i.e. whose actions are opaque and inexplicable.
>
> If we are to avoid a succession of Three-Mile- Island-type disasters or worse, then our expert systems must be open to interrogation and inspection. In short, a reasoning method that cannot be explained to a person is unsatisfactory, **even if it performs better than a human expert.**" [9, page 14]

The problems associated with the development of erroneous, unreliable, and unmaintainable software systems is one of the most serious problems in the field of computer science. The entire Software Engineering discipline is dedicated to solving this problem. ESs that do not have a capability for *interrogation and inspection* will only aggravate and complicate the problems Software Engineering is trying to solve. Thus, EFs provide an invaluable debugging tool for knowledge engineers and system designers.

The third reason EFs are important can almost be viewed as a by-product of the previous two capabilities. If EFs can be used by technical experts to establish credibility and confidence in the design and functionality of ESs, then it stands to reason that they can also be used as a teaching/tutorial aide for less knowledgeable users of these ESs.

These are the major reasons EFs are considered to be such important components of ESs. Let's now look at the three functions that must be considered in order to produce them.

Functional Framework

There are three high-level functions that constitute the basic framework in which all work on explanation generation is currently being done. Interestingly, these three functions are identified in the literature in several different ways. In this paper we will refer to them as Functions 1, 2, and 3. A high-level descrip-

tion of each will now be presented.

Function 1: The content of any explanation is based on the ES's internal examination (introspection) of its own problem-solving mechanism or behavior. Function 1 concerns itself with identifying ways to model the contents of this problem-solving mechanism (the knowledge and reasoning process of the system). In order to do this the knowledge and reasoning process must be represented in well defined, well structured methods or formalisms. In addition, these methods or formalisms must be appropriate for the specific problem-solving task being addressed and must be able to be examined by the system. Attacking this modeling process from the *knowledge and reasoning process* level is to approach it from a very high-level, abstract point of view. By identifying three different types of explanation that can be produced from the *knowledge and reasoning process* of an ES, Chandrasekaran et al. [4,5] have provided a more detailed level of abstraction from which to attack this problem. These three types of explanation are now described.

1. **Type 1** explanations are concerned with explaining why certain decisions were or were not made during the execution (runtime) of the system. These explanations use information about the relationships that exist between pieces of data and the knowledge (sets of rules for example) available for making specific decisions or choices based on this data. For example, Rule X fired because Data Y was found to be true.
2. **Type 2** explanations are concerned with explaining the knowledge base elements themselves. In order to do this, explanations of this type must look at *knowledge about knowledge*. For example, knowledge may exist about a rule that identifies this rule (this piece of knowledge) as being applicable ninety percent of the time. A type 2 explanation could use this information (this knowledge about knowledge) to justify the use of this rule. Other knowledge used in providing this type of explanation consists of knowledge that is used to develop the ES but which does not effect the operation of the system. This type of knowledge is referred to as **deep** knowledge.
3. **Type 3** explanations are concerned with explaining the runtime control strategy used to solve a particular problem. For example, explaining why one particular rule (or set of rules) was **fired** before some other rule is an explanation about the control strategy of the system. Explaining why a certain question (or type of question) was asked of the user in lieu of some other logical or related choice is another example. Therefore, type 3 explanations are concerned with explaining how and why the system uses its knowledge the way it does, a task that also requires the use of **deep** knowledge in many cases.

Function 2: This function concerns itself with providing an explanation to the user based on that user's particular needs and abilities. Every user is different. Each has a different level of understanding about the problem domain. Each has a different reason for wanting a particular explanation. Based on these differences, it may not be necessary for all available information about an explanation to be provided. Function 2 therefore, is concerned with determining ways to tailor explanations for individual users.

Function 3: This function concerns itself with **how** to convey or present the information to the user. Should natural language be used or will source code statements suffice? What about graphical displays? Should text and graphics be combined? These are the types of questions (or concerns) this function must consider.

Having now established a basic foundation of knowledge concerning explanation generation in ESs, lets look at the design and implementation of a specific EF.

3 EFFESS: Background Information

An ongoing research and development effort is being conducted at Wright State University (WSU) into the use of two important (and somewhat conflicting) technologies: *software engineering with Ada* and *AI applications*. The remainder of this paper discusses the design and implementation of an EF project (EFFESS) that is part of this WSU work.

The ES Shell

The ES shell being used in EFFESS was developed by Capt James Cardow as part of his master's degree requirements. [3] Cardow's shell was designed using the Object Oriented Programming (OOP) design methodology, partly because Ada supports this methodology and partly because OOP is closely related to the *frame-based* KRS Cardow chose to use. Reasoning within the system is performed by using either a *forward-chaining (data driven)* process, a *backward-chaining (hypothesis driven)* process, or a combination of both of these processes. Information in the system is stored in a *hierarchy of frames* and can be passed between these frames by using *inheritance* or by passing *messages*. *Demons* are attached to the slots of the frames and represent the execution mechanism of the system. Each type of demon performs a specific function on the slot to which it is attached and accomplishes this function by processing or **firing** an associated set of *production rule(s)*.

Initial Requirements

There are two major requirements for EFFESS that define the scope of this project. These are: the use of Ada as the implementation language for the project and the decision to **add** an EF to an existing system rather than **include** an EF in the design/redesign of an ES.

The need to include sound software engineering practices in the development of **any** computer software system is a well accepted fact and has been discussed elsewhere. [3] The requirement to **add** an EF rather than **include** its requirements and specifications in the design or redesign of an ES developed from three different sources. The first deals with the need for a *plug-in* type EF that provides a testing and debugging type explanation capability for ESs that are ultimately going to be embedded into larger systems of some kind.[10, page 70] The second is identified by Wick and Slagle, in their work on JOE [12]. They recognized that many operational ESs (ones they called *practice systems*) needed an EF but could not afford to go through a major redesign in order to acquire some of the sophisticated explanation capabilities being developed. They also recognized that an *effective* EF could be provided without going through this redesign process. Therefore, their effort was concentrated on providing effective EFs for on-line ESs **without making major modifications to the original ES code**, an effort that was directly applicable to this project. The third contributing source to this requirement was the *frame-based* KRS Cardow chose to use in the ES shell. After examining Cardow's shell, definite support existed for viewing this requirement as a feasible one.

Borrowed Ideas And Concepts

An extensive literature review was conducted as preparation for the EFFESS project. This subsection identifies those concepts and ideas that have been taken from the literature and used (in one way or another) in EFFESS. (See [10] for detailed discussions of these different EF research and development efforts.)

- *MYCIN* : The basic functionality of MYCIN's Reasoning Status Checker (RSC) is used in EFFESS. In addition, the two example questions listed for the RSC are implemented. While the functionality of the General Question and Answerer (GQA) was considered, its requirement for natural language processing was determined to be beyond the scope of this project. However, access to the different types of information required by the GQA is made available in EFFESS.

- *TEIRESIAS* : The four step process Davis identifies as being necessary to design an EF was used. Section 5 presents a discussion of EFFESS' design process using these four steps.

- *GUIDON* : While specific use of *meta-rules* was not needed in EFFESS, *meta-knowledge* concerning the implicit inheritance control mechanism of the frame-based KRS was used.

- *NEOMYCIN* : The primary focus of NEOMYCIN was on explaining diagnostic strategies. While EFFESS does provide a limited capability in this area, little of the work done in NEOMYCIN was used.

- *Explicit Development Models* : Most of Swartout's work involves the capturing and explanation of *deep* knowledge. Two aspects of EFFESS can be traced back to Swartout's work. First, the production rules in the system were encoded using descriptive, English-like names. This is a powerful feature of Ada and allows EFFESS to produce very descriptive explanations without having to concern itself with English translations of rules, use of text generators, etc. Swartout used a similar capability in the Digitalis Therapy Advisor. Second, Swartout recognized the software engineering assistance EFs can provide.

- *BLAII* : None of Weiner's work in BLAII is used in EFFESS because major changes to the ES shell would be required. However, a discussion of his *system view* and his *user view* with respect to EFFESS is presented in Section 6.

- *CLEAR* : Rubinoff's CLEAR system was designed to be an attachable front-end to an **independently developed** ES. While none of the functionality of CLEAR is included in EFFESS, the basic design goal of CLEAR is one of the primary objectives of this project.

- JOE : Wick and Slagle's work on JOE has been, by far, the most influential on this project. Four of the six functions they provide in JOE are implemented in EFFESS: WHAT, WHERE, WHY, and HOW. (However, these four functions are not presented in three different tenses, as was done in JOE.)

While we are discussing the literature information that has influenced the development of EFFESS, one final reference needs to be mentioned. In one section of their article, Building Knowledge-Based Systems with Procedural Languages [2], Butler, Hodil, and Richardson discuss EFs. In their discussion they identify three functions that an EF should perform: the Rule Query, the Why Query, and the Explain (or How) Query. Additionally, they identify the use of a *stack* as an appropriate data structure for maintaining a trace of the system's performance. All of this information is included in the functionality of EFFESS. However, the implementation of the Why Query and the How Query vary from the descriptions Butler, Hodil, and Richardson provide for these functions. EFFESS's How and Why functions are based on MYCIN's interpretation of what these queries mean. The reason for this difference stems from the fact that both systems are avoiding the problems of natural language processing by explicitly defining what is meant by Why and How.

4 EFFESS: A Frame-Based System

One of the most important aspects of our project is that it involves the explanation of a *frame-based* KRS. Therefore, to understand the specifics of this implementation effort, a detailed examination of this KRS is required. Interestingly, there are several differing opinions within the AI community concerning the definition of a *frame-based* KRS or *frame-based* system. While understanding these differences is not important for our discussion, a brief look at the origin of this KRS is needed.

There are two fundamental KRSs from which a *frame-based* KRS is derived, a *rule-based* KRS and a *frame* KRS. However, with respect to explanation generation, each of these KRSs has a major limitation. Rule-based systems can't define terms, describe objects, or identify static relationships among objects. Frame systems can't declaratively describe how to process the knowledge they contain. [7] However, by combining these two KRSs, the strength(s) of one can be used to overcome the weakness(s) of the other thereby creating a much more powerful and robust KRS. This new KRS is what we are calling a *frame-based* KRS and is the one used by Cardow in his ES shell.

There are several contributions that this *hybrid* KRS makes to explanation generation.

1. Because the production rules are attached to the slots via demons, a logical partitioning of the rule set is provided. Thus, a partial explanation as to the purpose of a given rule can be provided simply by examining the slot information to which it is attached.

2. The hierarchical structure identifies the relationships among objects and allows for an explanatory description of an object by simply identifying the sub-frames attached to it. Further explanation of each of the sub-frames can also be provided by identifying the slots attached to each sub-frame.

3. The implicit control/inferencing mechanism (*inheritance*) in this system is available for explanation and thus provides some Function 1, Type 3 explanation capabilities. In addition, slot values that were determined by inheritance can also be identified. This additional information enhances the explanation of that particular slot.

4. The hierarchical structure also provides a logical partitioning of the knowledge. Therefore, if an EF were to be enhanced by providing **deep** descriptive knowledge for an object, this information could be easily added as a slot to the appropriate frame or as an attribute to the appropriate slot.

5. As already noted by Firebaugh, this type of KRS provides

an easy, efficient process for handling queries about the KB because the knowledge is so well structured and easy to find.

6. This KRS provides two *typical* representations of expert knowledge (if — then rules, and object decomposition) that map almost directly to the way this knowledge is thought of by experts in the real world. Therefore, explanations of these objects should provide more understandable explanations simply because of their realistic representations in the system.

5 EFFESS : Design & Functionality

The Design

In deciding how to go about designing EFFESS, two concepts or ideas were used. The first was Davis' design process for EFs. [6, page 264] The second was the OOP methodology used by Cardow in designing the original ES shell.

The primitive operation chosen for EFFESS is the same as the one Davis chose for TEIRESIAS, the *invocation* or *firing* of a rule. While the execution of the *demons* in our frame-based system is identified as providing the execution control of the system, demons are not the most primitive operation. The specific operation a particular demon is identified to perform is carried out by firing the set of rule(s) associated with that demon. Therefore, the individual production rule is the primitive operation in EFFESS.

The execution trace in EFFESS is a stack data structure. As identified in Section 3, this idea was taken from Butler, Hodil, and Richardson's article on using procedural languages to build knowledge-based systems. A stack provides a straightforward mechanism for properly recording the order in which the rules were fired.

The global framework in which the execution trace can be understood in EFFESS is its frame-based KRS. Davis chose a *goal tree* for his work in TEIRESIAS because of the backward-chaining control structure used in MYCIN. While our system also provides a backward-chaining control structure (and a forward-chaining one as well), the framework for understanding the execution trace (for understanding why a rule was tested/fired at a particular time during the execution of the system) involves information contained in the KRS. Remember that information is passed between frames in one of two ways (through inheritance or by passing messages) and that demons are executed when a slot's value is requested but does not exist. Therefore, understanding why a rule was tested/fired at a particular time (as indicated by the execution trace) requires that we know the method that was used in attempting to obtain a value for the slot. This information is contained in the KRS.

The final step of Davis's EF design process (writing a program to explain the trace to a user) has been expanded in EFFESS. This is because several of the explanation functions of EFFESS don't use the execution trace in providing their explanations. They are able to get the information they need directly from the KRS. Therefore, the EFFESS program contains more than just the code related to explaining the execution trace to the user.

The OOP design methodology presented by Booch [1, Chapter 4] was used in EFFESS' design at three different levels of abstraction. A discussion of its use at the highest level of abstraction (the *complete ES Development Environment* level) is presented here. EFFESS and the ES shell are the two primary objects that have to interact in order to provide the functionality of this larger system. As the ES shell was designed using OOP (before EFFESS existed), the only change that needed to be made was in its visibility. The ES shell has to have access to (be able to see) EFFESS in order to build the execution trace. It also has to have access to EFFESS's processing entry procedure in order to pass control when it is time to generate explanations. Because of Ada's strong support of OOP, providing this visibility for the ES shell was easily done.

With respect to using OOP in designing EFFESS, the operations identified for this object are the explanatory functions it must perform and are described in the next section. The visibility EFFESS requires consists primarily of the ES shell's processing package and its KRS. (Visibility was also established to a useful package of I/O routines. However, this was done from a code reusability standpoint and was not functionally required). As for establishing EFFESS's external interface, the routines it uses to build its execution trace and its processing entry procedures have to be made available to the ES shell.

Functionality

In looking at the functional requirements of EFFESS, two different explanation environments were identified: the *runtime* environment and the *end-of-processing* environment. Interestingly, the explanation requirements for these two environments are not the same. While they do share several of the same requirements, each has some unique requirements as well. For example, during the *runtime* environment the user may be prompted for information needed by the system. Being able to explain WHY the system needs this information is an important explanation requirement in these situations. However, the *end-of-processing* environment has no reason to prompt the user for information, therefore no requirement exists for a WHY explanation capability. Conversely, once the system has completed its processing and arrived at some kind of decision, being able to SHOW the *critical decision path* (the sequence of rules that fired) the system used to arrive at this decision provides a great deal of important information. However, during system execution, explaining the *current decision* path is the important issue. Therefore, in support of these differences, EFFESS provides a set of *runtime explanation functions* and a set of *end-of-processing explanation functions.*

Runtime Functions

1. *Explain Rule* : The Explain Rule function explains the identified rule by displaying its contents to the screen. As descriptive, English-like naming conventions were used in the construction of the rules, no additional text generation is required to present an understandable explanation.

2. *Explain Why* : The Explain Why function is interpreted to mean, **Why is this information being requested?** and is presented to the user as an option whenever the user is prompted for information. The basis for providing this explanation is twofold. First, some component in the system (i.e. a rule) needs this information in order to continue processing. Second, this value is currently unknown and couldn't be determined via inheritance or message passing. Therefore, EFFESS's WHY explanation identifies the slot whose value is being requested, the frame to which this slot is attached, and the system component that is waiting on this slot in order to continue processing.

3. *Explain How* : The Explain How function is interpreted to mean, **How did the system arrive at this point in**

its processing? and uses the execution trace to provide this explanation. This function starts with the first rule tested by the system and recurses through the execution trace for as long as the user determines is necessary. One rule from the trace is explained for each successive HOW request the user provides. He can examine the entire execution trace (from the start rule all the way to the current rule being processed) or he can stop at whatever level he is comfortable/satisfied with. The explanation content for each rule is based on the specific type of rule it is. For example, rules associated with GOAL frames are chosen for execution because they represent the specific goal to be achieved. Therefore, EFFESS's HOW function explains these rules in this context, identifying the rule number, the goal to be achieved, etc. On the other hand, rules that prompt the user for information are chosen for execution because the system needs this information to continue processing. Therefore, these rules are explained with respect to the system component that is dependent on them for the needed information.

4. *Explain What* : The Explain What function is interpreted to mean, **What is the value of slot X?** and is available for use throughout the execution of the system. The user must provide the name of an attribute (slot) and will receive its corresponding value (or a *not found* error message) in return. This function represents one type of query function Firebaugh identified as one of the features easily supported by a frame-based system.

5. *Explain Where* : The Explain Where function is interpreted to mean, **Where is X located in the KRS?** and is used to explain the relationships of objects and attributes in the KB. The user must provide an object name or an attribute name and will receive an explanation of who/what the input item is related too. If the input item is an object, any inheritance relationships to other objects are identified. If the input item is an attribute, the object to which it is attached is identified. This function represents a second type of query function.

End-Of-Processing Functions

1. *Explain Rule, What, Where* : same as in runtime functions

2. *Show Critical Path* : The execution trace is used to provide this function. As the trace maintains the correct ordering of all rules tested during the system's execution, the critical path list is provided by simply looping through the execution trace and printing out the rule numbers of the rules whose *fired* flag has been set.

3. *Show Execution Trace* : This function also uses the execution trace to accomplish it function. Its purpose is to display the entire sequence of rules that were tested during the system's execution. However, a rule can be in one of three states during various stages of the system's execution: *pending* (awaiting information), *fired* (its antecedent tested true), or *failed* (its antecedent tested false). Therefore, this function identifies which state a particular rule is in at the various stages of its execution.

Functional Framework

In analyzing EFFESS with respect to the Functional Framework we find that Function 1, Type 1 explanations are provided by the HOW function, the SHOW Critical Path function, and the SHOW Execution Trace function. By using a stack to capture the execution trace of the system (the order in which the rules were accessed), the basic decision process of the system is identified. Type 2 explanations are provided by the Rule, What, Where, and Why functions, all of which provide information about the elements (objects and attributes) in the knowledge base. Function 1, Type 3 explanations are also provided to a limited degree. The Why function can identify certain instances when inheritance or message passing has been used.

EFFESS provides limited Function 2 capabilities by allowing the user to determine the degree of explanation provided by the HOW function. However, as the primary users of EFFESS are assumed to be system designers and knowledge engineers (based on the previously discussed need for *plug-in* test and debugging type EFs), the explanations provided in this system have been directly geared for these users.

With respect to Function 3 capabilities, EFFESS does provide English-like output due to the descriptive naming conventions used in the system. However, considerable room for improvement still exists in this area.

6 EFFESS: Conclusions

A Straightforward Development Process

Using the OOP design methodology provided several contributions to the development process. Probably the most influential is the fact that in analyzing the ES shell and the desire to add an EF to it (from an OOP point of view), enough of a feasible solution was identified to make the decision to attempt the effort in the first place. Additionally, the required visibility and interface between EFFESS and the ES shell were easily identified using this design methodology.

Closely coupled with the contributions OOP provided to the design of EFFESS were the contributions Ada provided in support of these OOP decisions. Due to the already identified relationship that exists between Ada and OOP, once the visibility and interface specifications had been established, implementing them in Ada was a straightforward process using the built in contructs it specifically provides for these purposes (i.e. *with* clauses, *package specifications*, *separate compilation units*, etc.).

With respect to the frame-based KRS used in the ES shell, many of the contributions this structure provides for explanation generation were described in Section 4. These contributions are universal in nature, in that they are provided by a frame-based KRS implemented in any language. However, a frame-based KRS implemented in Ada provides an additional contribution to explanation generation as a result of its **strong typing** requirements. While these requirements are being identified as a major contribution to the development of EFFESS, it is interesting to note that these same requirements were identified as a major obstacle in the development of the ES shell (see [3, Chapter 3] for complete details). In any case, Ada's strong typing required the different objects of the ES shell to be decomposed into different structures of nested records and record pointers in order to be implemented. The contribution this makes to EFFESS is that each of the decomposed parts of an object are available for explanation. Therefore, in a frame-based KRS implemented in Ada, a hierarchical decomposition of objects at two different levels of abstraction are available for explanation. At the *knowledge level* this hierarchy is provided by the KRS. At the *implementation level* this hierarchy is provided by the different structures of

nested records and record pointers mentioned above.

A Plug-In & Unplug Type EF

In evaluating how successful we were in accomplishing our *plug-in* EF objective, two answers are required. Overall, the effort appears to be a success. Absolutely none of the original ES shell data structures were changed at all and the processing routines were only changed (added to) in three places: in the routine where the execution trace had to be built, in the routine that interfaced with the user so as to gain access to EFFESS, and in the driver routine to provide the end of processing explanation capabilities. Due to Ada's *separate compilation* construct, all of EFFESS' code is contained in one Ada package. Therefore, to *unplug* EFFESS from the ES shell requires commenting out or removing fifteen lines of code in three different routines, removing the EF package from the compilation order, and recompiling the ES shell code.

From an explanation standpoint we can also consider EFFESS a success in that effective, useful explanations are provided, especially with regards to Type 1 and Type 2 explanations. However, we can also consider EFFESS to be only marginally successful because of its limited ability to produce Type 3 explanations. While this is true, it is important to note that the reason Type 3 explanations are limited **is not** due to a lack of explainable information but rather because we restricted ourselves to not changing any (or as little as possible) of the ES shell code. Had this restriction not existed, several Type 3 explanations could have been provided because information concerning inheritance, message passing, and the forward/backward chaining control mechanisms is available to be explained.

Cost versus Capability

In many ways, the entire EFFESS project has re-addressed or re-focused our attention on a very old and important issue in the use of any new technology, *cost* versus *capability*. Only finite amounts of resources (time, money, and manpower) exist for any given project. Often times, the cost of incorporating the most *state-of-the-art* capabilities a technology provides exceeds these finite limits. Therefore, efforts at identifying ways to provide **the most capability for the least amount of resources** are not only justified, but severely needed. This project has been one of these efforts. However, it is important to realize that a frame-based ES provides the potential for producing a much richer explanation capability than provided in this project, if one includes the requirements and specifications of an EF in the initial design stages of the ES.

Acknowledgment

This work was sponsored in part by WRDC/AAWA-1 of the U.S. Air Force, WPAFB, Ohio.

References

[1] Grady Booch. *Software Engineering with Ada.* The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, second edition, 1983.

[2] C. W. Butler, E. D. Hodil, and G. L. Richardson. Building knowledge-based systems with procedural languages. *IEEE Expert Magazine*, pages 47–59, Summer 1988.

[3] James C. Cardow. Toward an expert system shell for a common ada programming support environnment. Master's thesis, Wright State University, Dayton, Ohio, 1989.

[4] B. Chandrasekaran, J. Josephson, and Michael C. Tanner. Explaining control strategies in problem solving. *IEEE Expert Magazine*, pages 9–24, Spring 1989.

[5] B. Chandrasekaran, John R. Josephson, and A. Keuneke. Functional representation as a basis for generating explanations. Technical Research Report 86-BC-FUNEXPL, The Ohio State University Department of Computer and Information Science Laboratory for Artificial Intelligence Research, 1986.

[6] R. Davis and D. Lenat. *Knowledge-Based Systems In Artificial Intelligence.* McGraw-Hill, New York, New York, 1982.

[7] Richard Fikes and Tom Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, Sep 1985.

[8] Morris W. Firebaugh. *ARTIFICIAL INTELLIGENCE: A Knowledge-Based Approach.* Boyd and Fraser Publishing Co., Boston, Massachusetts, 1988.

[9] Richard Forsyth, editor. *Expert Systems: Principles and Case Studies.* Chapman and Hall Publishing Co., London, England, 1984.

[10] Vance M. Saunders. Explanation generation in expert systems (a literature review and implementation). Master's thesis, Wright State University, Dayton, Ohio, 1989.

[11] Roger C. Schank. *Explanation Patterns: Understanding Mechanically and Creatively.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1986.

[12] James R. Slagle and Michael R. Wick. An explanation facility for today's expert systems. *IEEE Expert Magazine*, pages 26–36, Spring 1989.

[13] Jess Stein, editor. *The Random House College Dictionary.* Random House, Inc., New York, New York, revised edition, 1979.